Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 343 (2005) 509–528

Theoretical Computer Science

www.elsevier.com/locate/tcs

Slicing techniques for verification re-use☆

Heike Wehrheim

Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany

Abstract

In this paper we discuss which properties of a formally verified component are preserved when the component is changed due to an adaption to a new use. More specifically, we will investigate when a temporal logic property of an Object-Z class is preserved under a modification or extension of the class with new features. To this end, we use the *slicing* technique from program analysis which provides us with a representation of the dependencies within the class in the form of a *program dependence graph*. This graph can be used to determine the effect of a change to the class's behaviour and thus to the validity of a temporal logic formula.

Keywords: Verification; Slicing; Temporal logic; Object-Z

1. Introduction

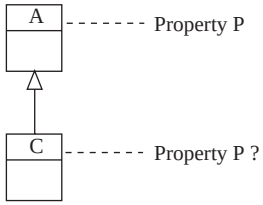
With the advent of component-based software engineering systems are more and more built from pre-fabricated components which are taken from libraries, adapted to new needs and assembled into a system. Furthermore, for the design of dependable systems formal methods are employed during the construction process to improve the degree of correctness and reliability. The combination of these two techniques—component-based design and formal methods—in system construction poses a large number of new research challenges that are under active investigation (see for instance the conference series on Formal Methods for Components and Objects).

☆ This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

E-mail address: wehrheim@uni-paderborn.de.

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2005.06.020

This paper studies one aspect arising in this area, based on the following scenario of a component-based construction. We assume that we have a library of components which are formally specified and proven correct with respect to certain requirements. During system construction components are taken from the library and (since they might not fully comply to its new use) are modified or even extended with new features. The question is then whether the proven properties are preserved under this specialisation and thus, whether we can also get a *re-use* of verification results and not just of components. More specifically, given a component *A* (which will be a single class here) and its modification or extension *C*, we are interested in knowing whether a property *P* holding for *A* still holds for *C* (see the following figure).



Although the picture might suggest that the relationship between *A* and *C* is that of inheritance (since we use the specialisation arrow of UML) we are actually interested in a more general relationship: *C* may be any class which is constructed out of *A*, may it be by inheritance or by a simple change of the existing specification.

As a first observation, it can be remarked that even a restriction to inheritance cannot ensure that properties are preserved: a subclass may differ from its superclass in any aspect and thus none of the properties holding for *A* might be preserved in *C*. Still, preservation of properties to subclasses is an important and intensively studied topic. Within the area of program verification, especially of Java programs, this question has already been tackled by a number of researchers [14,21,13]. In these approaches correctness properties are mainly formulated in Hoare logic, and the aim is to find proof rules which help to deduce subclass properties from superclass properties. In order to get correctness of these rules it is required that the subclass is a *behavioural subtype* [16] of the superclass. This assumption is also the basis of [26] which studies preservation of properties in an event-based setting with correctness requirements formulated as CSP processes.

In this paper we lift this assumption (although also looking at subtypes as a special case) and consider arbitrary classes constructed out of existing classes. For convenience we will often say that the class *C* is derived from *A*. Instead of employing restrictions on the derived class (in order to preserve properties) we will *compute* whether a property is preserved or might potentially be invalidated. This computation does not involve re-verification of the property but can be carried out on a special representation of the classes called *program dependence graphs*. Program dependence graphs carry all information about the dependencies within programs (or in our case, specifications) and thus can be used to determine the influence of a change or extension on proven properties. This technique originally comes from program analysis, where *slicing* techniques operating on program dependence graphs are used to reduce a program with respect to certain

variables of interest. Slicing techniques (or a similar technique called cone-of-influence reduction) are also being applied in software and hardware model checking for *reducing* programs [9,18,4].

In our framework classes are not written in a programming language but are defined in a state-based object-oriented formal method (Object-Z [22,5]). Correctness requirements on classes are formalised in a temporal logic (LTL [17]). As changes (specialisation) we allow the addition of attributes, the modification of existing methods and the extension with new methods. A comparable study about inheritance of CTL properties is described in [27], however, not employing the program dependence graphs of slicing which we use here and which allow for a more comprehensible representation of the dependencies within specifications.

The work presented here is a first step towards the application of slicing technique in the verification of *integrated specification formalisms*. An integrated formalism combines two or more existing formal methods into one new formalism, with the purpose of allowing for a convenient specification of different *views* on a system. Such views may cover the data and operations of a system (as the formalism Object-Z used here is doing) but also the dynamic behaviour (ordering of operations) as well as timing constraints. The benefit of such an integration of different specification techniques is the possibility of supplying a designer with an adequate formalism for every such view. The integrated specification technique to which we eventually intend to apply slicing is CSP-OZ-DC [11], an integration of Object-Z with the process algebra CSP [10] (to describe dynamic behaviour) and the interval logic duration calculus [32] (to describe timing constraints). The choice for using program dependence graphs and slicing to determine the influence of modifications on the holding of temporal logic formulae is (besides reasons of comprehensability) influenced by this goal: in integrated specification formalisms a number of different forms of dependencies have to be taken care of (even more than traditionally appearing in dependence graphs for slicing), and these can best be formalised within such a graph structure.

The paper is structured as follows. In the next section we define the necessary background for our study. Section 3 studies property preservation for behavioural subtypes and Section 4 introduces slicing as a more general technique for computing preserved properties for arbitrary changes. Section 5 discusses fairness constraints on classes which have to be introduced to cover liveness properties. The last section concludes and discusses related work.

This work is an extended version of [28] including all proofs of theorems plus an additional section on fairness and liveness.

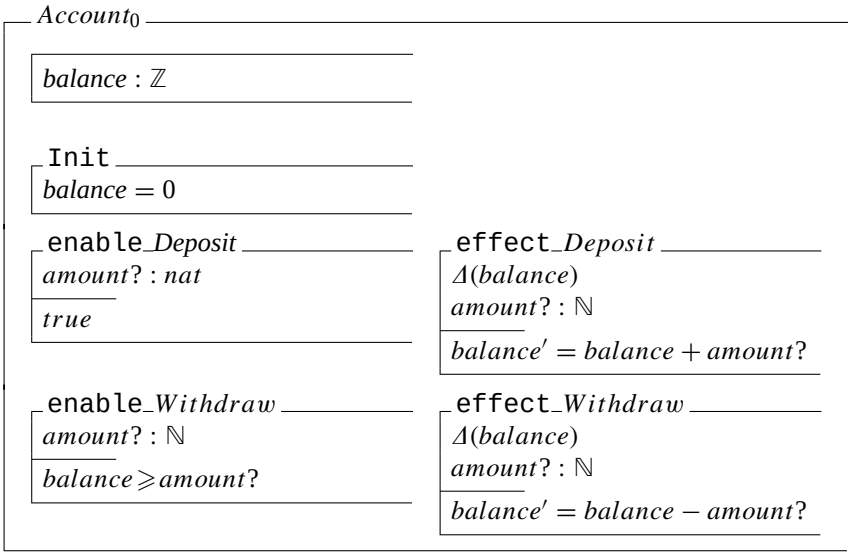
2. Background

This section describes the background necessary for understanding the results: the definition of classes in Object-Z, the temporal logic LTL and a result showing that LTL-X properties are preserved under stuttering equivalence. Stuttering equivalence will be used to compare class and derived class.

2.1. Class definitions

Classes are described in a formalism very close to Object-Z [22]. [1] Object-Z is an object-oriented extension of Z and thus a state-based specification technique.

The following specification of a simple account is the running example for our technique. It specifies the state of an account (with a certain *balance*), its initial value and two methods for depositing and withdrawing money from the account. Methods are specified with enable and effect schemas describing the guard (to the execution of) and the effect of executing the method. For instance, since the account may not be overdrawn, the guard of *Withdraw* specifies that the amount of money to be withdrawn may not exceed the balance (*amount*? is an input variable). The Δ-list of an effect schema fixes the set of variables which may be changed by an execution of the method.



In our definitions we use the following nongraphical formulation of classes. Classes consist of attributes (or variables) and methods to operate on attributes. Methods may have input parameters and may return values, referred to as output parameters. We assume variables and input/output parameters to have values from a global set D. A *valuation* of a set of variables V is a mapping from V to D. We let $R_V = \{\rho : V \to D\}$ stand for the set of all valuations of V; the set of valuations of input parameters *Inp* and output parameters *Out* can be similarly defined. For a valuation ρ of variables V we define $\rho\!\upharpoonright_{V'}$, $V' \subseteq V$ to be the function $\rho' : V' \to D$ such that $\rho'(v) = \rho(v)$ for all $v \in V'$. We assume that the

[1] In fact, it is the Object-Z part of CSP-OZ specifications [6], a formalism which integrates CSP with Object-Z. We use this formalism since we are ultimately interested in answering the question of property preservation for CSP-OZ.

initialisation schema precisely fixes the values of variables (i.e. is deterministic) in order to have just one initial state. [2]

A class is thus characterised by

- A set of *attributes* (or variables) V,
- an *initial valuation* of V to be used upon construction of objects: $I : V \to D$, and
- a set of methods (names) M with input and output parameters from a set of inputs *Inp* and a set of outputs *Out*. For simplicity we assume *Inp* and *Out* to be global. Each $m \in M$ has a guard $enable_m : R_V \times R_{\text{Inp}} \to \mathbb{B}$ ($\mathbb{B}$ is the booleans) and an effect $effect_m : R_V \times R_{\text{Inp}} \to R_V \times R_{\text{Out}}$. The guard specifies the states and inputs for which the method is executable and the effect determines the outcome of the method execution.

Note that we use the notation `enable_`m, `effect_`m when we refer to (the name of) the schema in the specification and $enable_m$, $effect_m$ when we refers to its semantics.

A class will thus be denoted by $(V, I, (enable_m)_{m\in M}, (effect_m)_{m\in M})$. We furthermore need to know the set of variables which are *set* and *referenced* by a schema: $Set(\texttt{enable\_}m) = \varnothing$, $Set(\texttt{effect\_}m)$ are the variables appearing in the Δ-list of the effect schema, and $Ref(\texttt{enable\_}m)$, $Ref(\texttt{effect\_}m)$ are those that syntactically appear in the schemas `enable_`m, `effect_`m, respectively. For `Init` we take $Set(\texttt{Init}) = V$ and $Ref(\texttt{Init}) = \varnothing$.

The semantics of a class is defined in terms of *Kripke structures*.

Definition 1. Let AP be a nonempty set of atomic propositions. A Kripke structure $K = (S, s_0, \to, L)$ over AP consists of a finite set of states S, an initial state $s_0 \in S$, a transition relation $\to \subseteq S \times S$ and a labelling function $L : S \to 2^{AP}$.

The set of atomic propositions determines what we may observe about a state. Essentially there are two kinds of properties we like to look at: the values of variables and the availability of methods. Thus the atomic propositions AP_A that we consider for a class $A = (V, I, (enable_m)_{m\in M}, (effect_m)_{m\in M})$ are

- $v\, op\, d$ for $v \in V$, $d \in D$ and op a symbol for a binary relation in D (e.g. $=, <, \ldots$),
- $enabled(m)$, $m \in M$,
- boolean combinations of these.

The Kripke structure semantics of a class definition is then defined as follows.

Definition 2. The semantics of (an object of) a class $A = (V, I, (enable_m)_{m\in M}, (effect_m)_{m\in M})$ is the Kripke structure $K = (S, s_0, \to, L)$ over AP_A with

- $S = R_V$,
- $s_0 = I$,
- $\to = \{(s, s') \mid \exists\, m \in M, \rho_{\text{in}} \in R_{\text{Inp}}, \rho_{\text{out}} \in R_{\text{Out}} : enable_m(s, \rho_{\text{in}}) \wedge effect_m(s, \rho_{\text{in}}) = (s', \rho_{\text{out}})\}$,
- $L(s) = \{p \in AP \mid s \vDash p\} \cup \{enabled(m) \mid \exists\, \rho_{\text{in}} \in R_{\text{Inp}} : enable_m(s, \rho_{\text{in}})\}$.

[2] This assumption is not essential but more convenient.

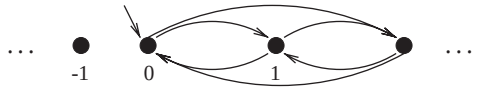


Fig. 1. Kripke structure of class $Account_0$.

We use the notation $s \vDash p$ for expressing that the atomic proposition p holds in state s. We write $s \xrightarrow{m} s'$ if execution of m leads from s to s'. Since the atomic propositions do not refer to inputs and outputs of methods, they are not reflected in the semantics. However, inputs and outputs can be embedded in the state and thus can be made part of the atomic propositions (see e.g. [23]).

Fig. 1 shows the Kripke structure (without L) of class $Account_0$. The numbers indicate the values of attribute *balance*. All states satisfying $balance < 0$ are unreachable. The upper arrows correspond to executions of *Deposit*, the lower to those of *Withdraw* (labels have been left out).

Furthermore, we have to fix the kind of changes allowed in derived classes. We do not allow to remove methods, but methods can be arbitrarily modified as well as new methods and variables be introduced.

Definition 3. Let A and C be classes. C is a specialisation of A if $V_A \subseteq V_C$, $M_A \subseteq M_C$ and $I_C|_{V_A} = I_A$.

2.2. LTL formulae

The temporal logic which we use for describing our properties on classes is linear-time temporal logic (LTL) [17].

Definition 4. The set of *LTL* formulae over *AP* is defined as the smallest set of formulae satisfying the following conditions:

- $p \in AP$ is a formula,
- if φ_1, φ_2 are formulae, so are $\neg\varphi_1$ and $\varphi_1 \vee \varphi_2$,
- if φ is a formula, so are $X\varphi$ (Next), $\Box\varphi$ (*Always*), $\Diamond\varphi$ (*Eventually*),
- if φ_1, φ_2 are formulae, so is $\varphi_1\ U\ \varphi_2$ (*Until*).

As usual, other boolean connectives can be derived from $\neg$ and $\vee$. The next-less part of LTL is referred to as LTL-X. LTL formulae are interpreted on *paths* of the Kripke structure, and a formula holds for the Kripke structure if it holds for all of its paths.

Definition 5. Let $K = (S, s, \rightarrow, L)$ be a Kripke structure. A finite or infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ is a path of K iff $s = s_0$ and $(s_i, s_{i+1}) \in \rightarrow$ for all $0 \leqslant i$. For a path $\pi = s_0 s_1 s_2 \ldots$ we write $\pi[i]$ to stand for s_i and π^i to stand for $s_i s_{i+1} s_{i+2} \ldots$. The length of a path π, $\#\pi$, is defined to be the number of states (in case of a finite path) or ∞ (in case of an infinite path).

Temporal logics (like LTL) are usually interpreted on infinite paths. We deviate from that here because objects may also exhibit finite behaviour: if no methods are called from the outside anymore, the object just stops. This has, however, consequences on the holding of liveness properties: since, for instance, s_0 alone is a path as well, a liveness property can only hold if it already holds in the initial state. Thus we essentially treat *safety* here. Liveness can be treated if we additionally make some fairness assumptions on the environment of an object which ensure progress. This will be discussed in Section 5.

Definition 6. Let $K = (S, s_0, \rightarrow, L)$ be a Kripke structure and φ an *LTL* formula, both over *AP*. *K* satisfies φ, $K \models \varphi$, iff $\pi \models \varphi$ holds for all paths π of *K*, where $\pi \models \varphi$ is defined as follows:

- $\pi \models p$ iff $p \in L(\pi[0])$,
- $\pi \models \neg\varphi$ iff not $\pi \models \varphi$,
- $\pi \models \varphi_1 \vee \varphi_2$ iff $\pi \models \varphi_1$ or $\pi \models \varphi_2$,
- $\pi \models X\ \varphi$ iff $\#\pi > 1 \wedge \pi^1 \models \varphi$,
- $\pi \models \Box\varphi$ iff $\forall\, i, 0 \leqslant i \leqslant \#\pi : \pi^i \models \varphi$,
- $\pi \models \Diamond\varphi$ iff $\exists\, i, 0 \leqslant i \leqslant \#\pi : \pi^i \models \varphi$,
- $\pi \models \varphi_1\ U\ \varphi_2$ iff $\exists\, k, 0 \leqslant k \leqslant \#\pi : \pi^k \models \varphi_2$ and $\forall\, j, 0 \leqslant j < k : \pi^j \models \varphi_1$.

For our bank example we for instance have the following properties. The Kripke structure K_{Account_0} of $Account_0$ fulfils

$$K_{\text{Account}_0} \models \Box(balance \geqslant 0)\ ,$$

$$K_{\text{Account}_0} \models \Box(enabled(Deposit)).$$

2.3. Stuttering equivalence

For showing that properties are preserved under change, or more particular, that a certain property still holds for a derived class, we will later compare both classes according to a notion of equivalence called *stuttering equivalence*. Stuttering equivalence is defined with respect to some set of atomic propositions and roughly says that as far as these propositions of interest are concerned two Kripke structures have an equivalent behaviour. All transitions changing propositions outside those of interest are regarded as stuttering steps.

Stuttering equivalence is first defined on paths and then lifted to Kripke structures. Intuitively, two paths are stuttering equivalent with respect to some set of atomic propositions *AP*, if they can be divided into blocks in which propositions from *AP* stay stable and the *i*th block in π has the same set of propositions as the *i*th block in ρ (illustrated in Fig. 2).

Thus, the paths may vary in the number of steps within a block but not in the atomic propositions in blocks *as far as the set AP is concerned*.

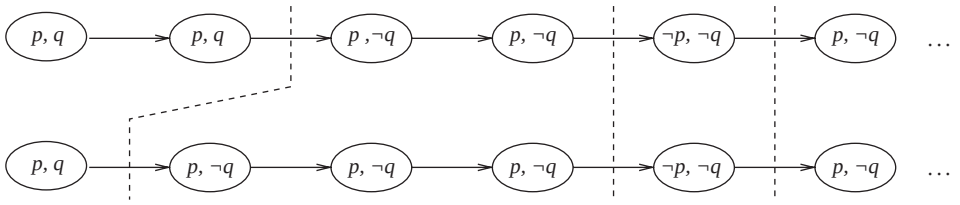


Fig. 2. Stuttering equivalent paths.

Definition 7. Two infinite paths $\pi = s_0 s_1 s_2 \ldots$ and $\rho = r_0 r_1 r_2 \ldots$ are stuttering equivalent wrt. a set of atomic propositions AP ($\pi \approx_{AP} \rho$) if there are two sequences of indices $0 = i_0 < i_1 < i_2 < \cdots$ and $0 = j_0 < j_1 < j_2 < \cdots$ such that for every $k \geqslant 0$

$$\begin{aligned} &L(s_{i_k}) \cap AP = L(s_{i_k+1}) \cap AP = \cdots = L(s_{i_{k+1}-1}) \cap AP \\ &\quad = L(r_{j_k}) \cap AP = L(r_{j_k+1}) \cap AP = \cdots = L(r_{j_{k+1}-1}) \cap AP \end{aligned}$$

A finite path $\pi = s_0 \ldots s_n$ is stuttering equivalent to an infinite path ρ if its extension with an infinite number of repetitions of the last state, i.e. $s_0 s_1 \ldots s_n s_n s_n \ldots$, is stuttering equivalent to σ. (And similarly for two finite paths.)

The last part of the definition guarantees that an infinite path can only be stuttering equivalent to a finite path if from some state on atomic propositions in AP do not change anymore.

Definition 8. Let $K_i = (S_i, s_{0,i}, \rightarrow_i, L_i)$, $i = 1, 2$, be Kripke structures over AP_1, AP_2, respectively. K_1 and K_2 are stuttering equivalent with respect to a set of atomic propositions $AP \subseteq AP_1 \cap AP_2$ ($K_1 \approx_{AP} K_2$) iff

- initial states agree on AP:

$$L_1(s_{0,1}) \cap AP = L_2(s_{0,2}) \cap AP,$$

- for each path π in K_1 starting from $s_{0,1}$ there exists a path π' in K_2 starting from $s_{0,2}$ such that $\pi \approx_{AP} \pi'$,
- and vice versa, for each path π in K_2 starting from $s_{0,2}$ there exists a path π' in K_1 starting from $s_{0,1}$ such that $\pi \approx_{AP} \pi'$.

Stuttering equivalent Kripke structures satisfy the same set of LTL-X properties [20,4]. The next operator has to be omitted since stuttering may introduce additional steps in one structure which have no counterpart in the other.

Theorem 1. *Let φ be an LTL-X formula over AP and K_1, K_2 Kripke structures. If $K_1 \approx_{AP} K_2$ then*

$$K_1 \vDash \varphi \quad \textit{iff} \quad K_2 \vDash \varphi.$$

3. Property preservation

Now that we have set the ground, we have another look at our example and make two changes to the class. The first is an *extension* of the class, we add one new method for balance checking. Here, we use inheritance to avoid having to write the whole specification again.

Account$_1$

inherit *Account*$_0$

enable_*CheckBalance*

$$true$$

effect_*CheckBalance*

$$bal! : \mathbb{Z}$$

$$bal! = balance$$

Here *bal*! is an output variable. The second change is a *modification*, we modify the account such that it allows overdrawing up to a certain amount. Here, we inherit all parts but the definition of *Withdraw* which is overwritten by the new definition.

Account$_2$

inherit *Account*$_0$

modifies *Withdraw*

$$overdraft : \mathbb{N}$$

Init

$$overdraft = 1000$$

enable_*Withdraw*

$$amount? : \mathbb{N}$$

$$balance - amount? \geqslant -overdraft$$

effect_*Withdraw*

$$\Delta(balance)$$

$$amount? : \mathbb{N}$$

$$balance' = balance - amount?$$

The question is then which of our properties are preserved, i.e. which of the following questions can be answered with yes.

$K_{\text{Account}_1} \models \Box(balance \geqslant 0)$?
$K_{\text{Account}_1} \models \Box(enabled(Deposit))$?
$K_{\text{Account}_2} \models \Box(balance \geqslant 0)$?
$K_{\text{Account}_2} \models \Box(enabled(Deposit))$?

For this simple example, the answers are easy. What we aim at is, however, a general technique which answers such questions. In general, the two changes made are of two

different types. The derived class can be a *behavioural subtype* of the original class (and then all properties are preserved) or not (and then a more sophisticated technique has to be applied to find out whether a property is preserved).

In this section, we deal with the first, more simple case. The second case is dealt with in the next section. A behavioural subtype can be seen as a conservative extension of a class: new methods may read but may not modify old variables.

Definition 9. Let A, C be two classes, C a specialisation of A. C is a behavioural subtype (*or short, subtype*) of A iff the following conditions hold:

- $\forall m \in M_C \setminus M_A$: $Set_C(\texttt{effect\_}m) \subseteq V_C \setminus V_A$
  (m only modifies new variables),
- $\forall m \in M_A : \texttt{enable\_}m^C = \texttt{enable\_}m^A \wedge \texttt{effect\_}m^C = \texttt{effect\_}m^A$
  (old methods not modified).

Subtypes inherit all properties as long as they are only talking about propositions over the old attributes and methods.

Theorem 2. *Let C, A be classes, C a behavioural subtype of A. Let furthermore AP_A be the set of atomic propositions over V_A and M_A. For all LTL-X formulae φ over AP_A we then have*

$$A \models \varphi \iff C \models \varphi.$$

The proof proceeds by showing that C and A are stuttering equivalent. The stuttering steps in C are those belonging to executions of the new methods: they do not change old attributes and thus do not affect AP.

In the proof we use an operator $\oplus$ on states: If $s : V_1 \to D$, $t : V_2 \to D$ are valuations and $V_1 \cap V_2 = \varnothing$ then $s \oplus t : V_1 \cup V_2 \to D$ is the combination of these two functions defined by $(s \oplus t)(v) = s(v)$ iff $v \in V_1$ and $t(v)$ iff $v \in V_2$.

Proof of Theorem 2. Let $K_A = (S_A, s_{0,A}, \to_A, L_A)$ and $K_C = (S_C, s_{0,C}, \to_C, L_C)$ be the Kripke structures of A and C, respectively. We have to show that K_A and K_C are stuttering equivalent wrt. AP_A. For this we use the following relation between states of K_A and K_C:

$$B = \{(s_A, s_C) \mid (s_A|_{V_A}) = (s_C|_{V_A})\}$$

As a first observation we get: $(s_A, s_C) \in B$ implies $L_A(s_A) \cap AP_A = L_C(s_C) \cap AP_A$. Since for specialisations we have required that $I_C|_{V_A} = I_A$ holds we get $(s_{0,A}, s_{0,C}) \in B$ and hence the first condition of stuttering equivalence holds.

Assume now that $\sigma = s_0 s_1 s_2 \ldots$ is a path in K_A. It can either be finite or infinite. We construct a corresponding path $\rho = t_0 t_1 t_2 \ldots$ in K_C that such $(s_i, t_i) \in B$ for all $0 \leqslant i$, and hence $\sigma \approx_{AP} \rho$.

- Set $t_0 = s_{0,C}$: since s_0 is the initial state of K_A it is related to t_0.

- Next take $(s_i, t_i) \in B$ and let m be the method which is executed in K_A to get from s_i to s_{i+1}. Since $m \in M_A$ and is hence not changed in C, m is also enabled in t_i. The state t_i can be decomposed into s_i and an assignment of values to the variables in $V_C \setminus V_A$: $t_i = s_i \oplus x_i$. Now take t_{i+1} to be $s_{i+1} \oplus x_i$. By construction $(s_{i+1}, t_{i+1}) \in B$.

Reverse direction: Assume $\rho = t_0 t_1 t_2 \ldots$ to be a path of K_C. We construct a sequence of states $\sigma = s_0 s_1 s_2 \ldots$. This sequence might not immediately be a path of K_A (since states may be repeated in the middle) but can be made into one by omitting repetitions. The state s_i is simply $t_i|_{V_A}$. By construction we hence have $(s_i, t_i) \in B$. Consequently, if a method of A is taken from t_i to t_{i+1} then it is also enabled in s_i and execution leads to s_{i+1}. If a method from $M_C \setminus M_A$ is taken then $s_i = s_{i+1}$ (by definition of subtypes). The path σ' of K_A is obtained by erasing all repeated states in the middle. □

Coming back to our example, *Account*$_1$ is a behavioural subtype of *Account*$_0$: *CheckBalance* only references *balance* but does not modify it. Hence both properties are preserved

$$K_{\text{Account}_1} \models \Box(\mathit{balance} \geqslant 0)$$
$$K_{\text{Account}_1} \models \Box(\mathit{enabled}(\mathit{Deposit})).$$

4. Slicing

In this section we look at the more general case, where the modifications do not lead to subtypes. For this case, we cannot get one general result but have to specifically look at the changes made and the properties under interest.

The technique we use for computing whether a property is preserved under a specific change is the *slicing* technique of program analysis [24]. In program analysis slicing is originally used for debugging and testing, and answers questions like the following: "given a variable v and a program point p, which part of the program may influence the value of v at p?". Here, we like to extract a similar kind of information about our changes: "given some propositions and some change, does it influence the value of these propositions?". Technically, slicing operates on graphs which contain information about the dependencies within a program, so called *program dependence graphs* (PDG). A similar graph is now built for Object-Z classes. It starts from the control flow graph (CFG) of a class (depicted in Fig. 3), which contains

- one node n_0 labelled *Init*,
- one node n_{DO} labelled *DO* (nondeterministic choice),
- for every method m two nodes n_{en_m} and n_{eff_m} labelled `enable_`m and `effect_`m.

We let $\rightarrow_{\text{CFG}}$ denote the arrows in this graph, i.e. the relation between nodes, and $\rightarrow^{+}_{\text{CFG}}$ its transitive closure. The program dependence graph is obtained from the CFG by erasing all arrows and adding new ones corresponding to the control and data dependencies of the class. Formally,

Definition 10. A program dependence graph (PDG) of a class specification $A = (V, I, (\texttt{enable\_}m)_{m \in M}, (\texttt{effect\_}m)_{m \in M})$ is a graph $G = (K, l, \rightsquigarrow, \rightarrowtail)$ with

- $K = \{n_0, n_{DO}\} \cup \{n_{\text{en}\_m} \mid m \in M\} \cup \{n_{\text{eff}\_m} \mid m \in M\}$ a set of nodes,

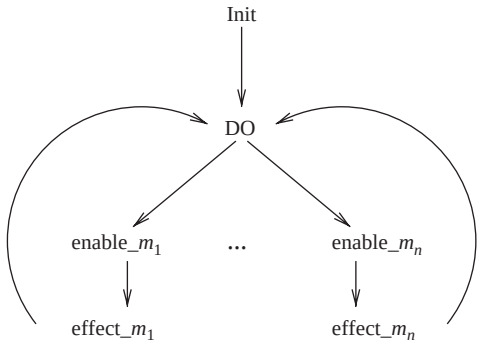


Fig. 3. Control flow graph of a class.

- l a labelling function with

$$\begin{aligned} l: \quad n_0 &\mapsto \mathit{Init} \\ n_{DO} &\mapsto \mathit{DO} \\ n_{\text{en\_}m} &\mapsto \texttt{enable\_}m \\ n_{\text{eff\_}m} &\mapsto \texttt{effect\_}m \end{aligned}$$

- $\rightsquigarrow \subseteq K \times K$ the data dependence edges defined by

$$n \rightsquigarrow n' \text{ iff } \exists x \in V : x \in \mathit{Set}(l(n)) \text{ and } x \in \mathit{Ref}(l(n')) \text{ and } n \rightarrow^{+}_{\text{CFG}} n',$$

- $\rightarrowtail \subseteq K \times K$ the control dependence edges defined by

$$n \rightarrowtail n' \text{ iff } \exists m \in M : l(n) = \texttt{enable\_}m \text{ and } l(n') = \texttt{effect\_}m\ .$$

Here, we take $\mathit{Set}(DO) = \mathit{Ref}(DO) = \varnothing$. For class $\mathit{Account}_0$ this gives rise to the graph shown in Fig. 4.

For computing whether a property of A is preserved in C, we build a PDG including methods and dependencies of *both* A and C. In this $\text{PDG}_{A,C}$ we next determine the *forward slice* of all modified or new methods. The forward slice of a set of nodes N is the part of the graph which is forward reachable from nodes in N via data or control dependencies.

Definition 11. Let C, A be classes, C a specialisation of A. Let furthermore N be the nodes belonging to methods which are changed or new in C, i.e.

$$\begin{aligned} N = \{n \mid\ & \big(l(n) \in \{\texttt{enable\_}m, \texttt{effect\_}m \mid m \in M_C \backslash M_A\}\big) \vee \\ & \big(\exists m \in M_A : l(n) = \texttt{enable\_}m \wedge \mathit{enable}^C_m \neq \mathit{enable}^A_m\big) \vee \\ & \big(\exists m \in M_A : l(n) = \texttt{effect\_}m \wedge \mathit{effect}^C_m \neq \mathit{effect}^A_m\big)\}. \end{aligned}$$

The forward slice of N is the set of nodes in $\text{PDG}_{A,C}$ which are forward reachable from N, i.e.

$$fs(N) = \{n' \in K \mid \exists n \in N : n(\rightsquigarrow \cup \rightarrowtail)^*_{\text{PDG}_{A,C}} n'\}.$$

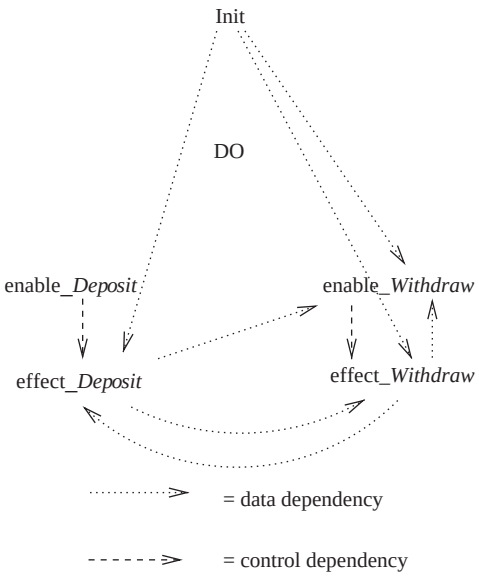


Fig. 4. PDG of class $Account_0$.

The forward slice of N is the part of the class which is directly or indirectly influenced by the changes. The atomic propositions appearing in this part might have their values changed. We let AP_N denote the atomic propositions over variables or methods in the forward slice of N (plus those over new variables, which might sometimes not be in $fs(N)$ since a variable might never be changed

$$\begin{aligned} AP_N = \{ & v \, op \, d \mid d \in D, op \text{ an operator } \wedge \\ & \big(v \in V_C \setminus V_A \vee \exists n \in fs(N) : v \in Set_C(l(n)) \cup Set_A(l(n))\big)\} \\ \cup \{ & enabled(m) \mid \exists n \in fs(N) : l(n) = \texttt{enable\_}m\}. \end{aligned}$$

Since these atomic propositions are potentially affected by the change, a formula talking about them might not hold in the derived class anymore. However, if a formula does not use propositions in AP_N then it is preserved.

Theorem 3. *Let A, C be classes, C a specialisation of A, and let N be the set of methods changed or new in C. If φ is an LTL-X formulae over $AP \setminus AP_N$, then the following holds*:

$$A \models \varphi \iff C \models \varphi \, .$$

The proof again proceeds by showing that K_A and K_C are stuttering equivalent wrt. $AP \setminus AP_N$. We first prove a lemma about methods. We let AP' stand for $AP \setminus AP_N$ and let V' be the subset of old variables not set by methods in $fs(N)$, $V_N = V \setminus V'$ its complement. Analogously M' is the set of methods not in $fs(N)$ and M_N those in $fs(N)$.

Lemma 1. *Let s, t be states of K_A, K_C, respectively, and assume $L_A(s) \cap AP' = L_C(t) \cap AP'$. Then the following holds*:
(1) $\forall m \notin N$, $\texttt{enable\_}m \notin fs(N)$:

$$m \text{ enabled in } s \Leftrightarrow m \text{ enabled in } t$$

(2) *If in addition* $\texttt{effect\_}m \notin fs(N)$ *and m is enabled in s* (*and hence in t*) *then*

$$s \xrightarrow{m} s', t \xrightarrow{m} t' \Rightarrow$$
$$L_A(s') \cap AP' = L_C(t') \cap AP'.$$

Proof.
(1) We have to show that $Ref(\texttt{enable\_}m) \subseteq V'$. Since s and t agree on variables in V' it follows that m is either enabled in both states or in none. Let n be the node labelled $\texttt{enable\_}m$. We assume the contrary

$$\exists v \in V_N : v \in Ref(\texttt{enable\_}m)$$
$$\Rightarrow \exists n' \in fs(N) : v \in Set(l(n'))$$
$$\Rightarrow \text{ there exists a data dependence edge } n' \rightarrow^* n$$
$$\Rightarrow n \in fs(N) (Contradiction)$$

(2) Let m be enabled in s and t. We have $Set(\texttt{effect\_}m) \subseteq Ref(\texttt{effect\_}m) \subseteq V'$. By the precondition of the lemma we can divide s and t in a V'-part and the rest: $s = s_{V'} \oplus s_{V_N}$ and $t = t_{V'} \oplus t_{V_N}$ such that $s_{V'} = t_{V'}$. Furthermore, execution of m modifies $Set(\texttt{effect\_}m)$ only and since $m \notin N$ it modifies it in the same way in s and t, that is

$$s_{V'} \oplus s_{V_N} \xrightarrow{m} s'_{V'} \oplus s_{V_N} = s'$$
$$s_{V'} \oplus t_{V_N} \xrightarrow{m} s'_{V'} \oplus t_{V_N} = t'$$

and hence $L_A(s') \cap AP' = L_C(t') \cap AP'$. □

Proof of Theorem 3. Let $K_A = (S_A, s_{0,A}, \rightarrow_A, L_A)$ and $K_C = (S_C, s_{0,C}, \rightarrow_C, L_C)$ be the Kripke structures of A and C, respectively. We have to show that K_A and K_C are stuttering equivalent wrt. AP'.

Initialisation: since specialisation required $I_C|_{V_A} = I_A$ and V' is a subset of V_A we get $L_A(s_{0,A}) \cap AP' = L_C(s_{0,C}) \cap AP'$.
(1) Let $\sigma = s_0 s_1 s_2 \ldots$ be a path of K_A. It can either be finite or infinite. We have to construct a path $\rho = t_0 t_1 t_2 \ldots$ in K_C such that $\sigma \approx_{AP'} \rho$. Following the proof of Theorem 2 we construct it in such a way that $L_A(s_i) \cap AP' = L_C(t_i) \cap AP'$ holds.
- Set $t_0 = s_{0,C}$ and we have already shown that the same set of propositions from AP' holds for s_0 and t_0.
- Take some (s_i, t_i) which has already been constructed and assume $s_i \xrightarrow{m} s_{i+1}$. There are several cases to consider now (note that due to the control dependence edges it cannot be the case that $\texttt{enable\_}m \in fs(N)$ but $\texttt{effect\_}m \notin fs(N)$):
  (a) $m \notin N \wedge \texttt{enable\_}m \notin fs(N) \wedge \texttt{effect\_}m \notin fs(N)$: By the previous lemma m is enabled in t_i and for $t_i \xrightarrow{m} t_{i+1}$ we have $L_A(s_{i+1}) \cap AP' = L_C(t_{i+1}) \cap AP'$.

(b) $m \notin N \wedge \texttt{enable\_}m \notin fs(N) \wedge \texttt{effect\_}m \in fs(N)$: Again by the same lemma m is enabled in t_i. Take t_{i+1} to be the state reached by executing m. Since $\texttt{effect\_}m \in fs(N)$ we have $Set(m) \subseteq V_N$ (by definition of V_N) and (since no variables of V_N are considered in AP') we get $L_A(s_{i+1}) \cap AP' = L_C(t_{i+1}) \cap AP'$.
(c) $m \notin N \wedge \texttt{enable\_}m \in fs(N) \wedge \texttt{effect\_}m \in fs(N)$: Then we set t_{i+1} to t_i. Again, since $Set(m) \subseteq V_N$ we get $L_A(s_{i+1}) \cap AP' = L_C(t_{i+1}) \cap AP'$.
(d) $m \in N$: same as last case (c).

Finally we erase duplications of states from ρ thereby getting a path of K_C. Note the following: IF σ is finite so is ρ. For infinite paths σ the constructed paths ρ can be either finite or infinite. Stuttering equivalence is nevertheless achieved in both cases.

(2) Converse direction: the construction of a path σ of K_A from a path ρ of K_C proceeds analogously. □

For our example, the PDG for $Account_0, Account_2$ is the same as those of $Account_0$. The set of changed methods N is {*Withdraw*}. Nodes not in the forward slice of *Withdraw* are {*Init*, *DO*, `enable_Deposit`}. The variable *balance* is set by a method in the forward slice, but `enable_Deposit` is not in the forward slice. Hence, concerning our properties, we know that one of them is preserved.

$$K_{\text{Account}_2} \models \Box(enabled(Deposit))$$

but for the question "$K_{\text{Account}_2} \models \Box(balance \geqslant 0)$?" our theorem does not tell us the answer (and in fact this property does not hold anymore).

The case of changes leading to subtypes can be seen as one particular instance of this more general result: for subtypes we know by definition that the forward slice (of the new methods) will only contain new methods and thus affects only new variables. Hence, the proof of Theorem 3 can be seen as an alternative way of proving Theorem 2.

The PDG of $Account_0, Account_1$ is depicted in Fig. 5. As can be seen, in the forward slice of *CheckBalance* there is only *CheckBalance*.

5. Liveness

So far, our approach has not covered liveness properties. This was due to the fact that we included finite paths into our interpretation of LTL formulae, and this was necessary since we cannot assume in general that methods of objects are called infinitely often. In this section we will lift this restriction by making additional assumptions on the environment of an object. These assumptions will be formulated as a set of methods that we assume to be called infinitely often. Technically such an assumption is a *fairness constraint* for a class (or its Kripke structure) and henceforth we only consider *fair paths* of Kripke structures.

Definition 12. A fair Kripke structure $K = (S, s_0, \rightarrow, L, F)$ is a Kripke structure (over a set of methods M) such that $\rightarrow \subseteq S \times M \times S$ and $F \subseteq M$ is a fairness constraint.

Since fairness is formulated on methods of a class we now take the methods explicitly into the transitions and consider paths as being sequences of states and operations.

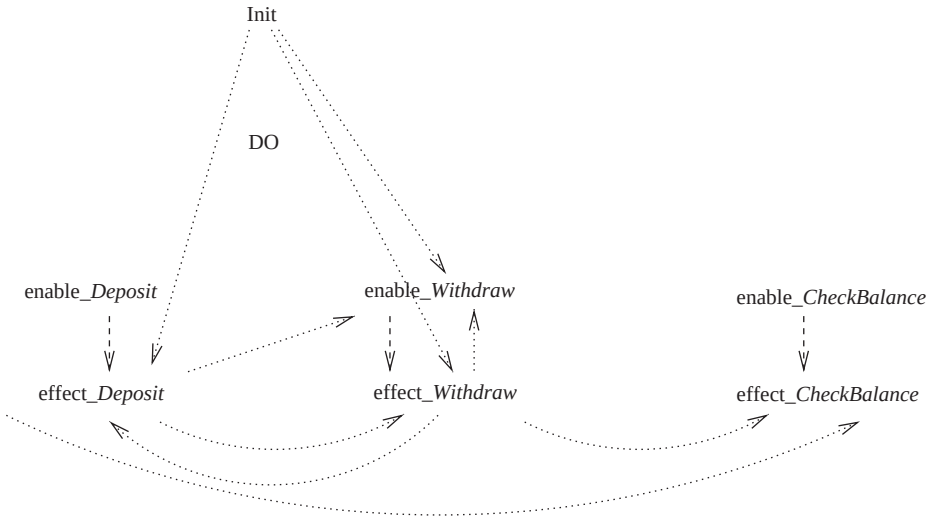


Fig. 5. PDG of $Account_0, Account_1$.

Definition 13. Let $K = (S, s, \rightarrow, L, F)$ be a fair Kripke structure. A finite or infinite sequence of states and operations $\pi = s_0 a_0 s_1 a_1 s_2 a_2 \ldots$ is a path of K iff $s = s_0$ and $s_i \xrightarrow{a_i} s_{i+1}$ for all $0 \leqslant i$. For a path $\pi = s_0 a_0 s_1 a_1 s_2 a_2 \ldots$ we define $inf(\pi)$ to be $\{a \mid a = a_i \text{for infinitely many } i\}$. A path is fair (with respect to F) iff $inf(\pi) \cap F \neq \varnothing$.

By definition, all fair paths are infinite. The interpretation of LTL formulae can then be restricted to fair paths.

Definition 14. Let $K = (S, s_0, \rightarrow, L, F)$ be a fair Kripke structure and φ an LTL formula. K fairly satisfies φ ($K \models_F \varphi$) iff $\pi \models \varphi$ holds for all fair paths π of K.

To preserve the fair interpretation of formulae stuttering equivalence has to be restricted to fair paths as following:

Definition 15. Let $K_i = (S_i, s_{0,i}, \rightarrow_i, L_i, F_i)$, $i = 1, 2$, be fair Kripke structures over AP_1, AP_2, respectively. K_1 and K_2 are fair stuttering equivalent with respect to a set of atomic propositions $AP \subseteq AP_1 \cap AP_2$ ($K_1 \approx^F_{AP} K_2$) iff

- initial states agree on AP:

$$L_1(s_{0,1}) \cap AP = L_2(s_{0,2}) \cap AP,$$

- for each fair path π in K_1 starting from $s_{0,1}$ there exists a fair path π' in K_2 starting from $s_{0,2}$ such that $\pi \approx_{AP} \pi'$,
- and vice versa, for each fair path π in K_2 starting from $s_{0,2}$ there exists a fair path π' in K_1 starting from $s_{0,1}$ such that $\pi \approx_{AP} \pi'$.

Since we have no particular knowledge about the environment of an object the most general form of fairness constraint is the whole set of methods. Then fairness only guarantees that paths are infinite. The question is what impact the fairness constraint has on the validity and preservation of properties. The restriction to fair paths leads to the validity of additional liveness properties that are established by calls to certain methods (which has not been guaranteed without fairness). In order to preserve these liveness properties we have to ensure that the same method(s) are being called in the derived class and, moreover, that they establish the same property, i.e. are unchanged. Thus, the derived class C inherits a property of the class A which is established using a fairness constraint $M \subseteq M_A$ if M is the fairness constraint for C and M is not part of $fs(N)$.

In the following theorem we use the notation concerning modified variables, methods and atomic propositions as proposed in the previous section.

Theorem 4. *Let A, C be classes, C a specialisation of A, and let $F \subseteq M_A$ be a fairness constraint. Let furthermore AP' and M' be as in Theorem* 3 (*the set of unchanged atomic propositions and unchanged methods, respectively*). *If φ is an LTL-X formula over AP' and $F \subseteq M'$ then*

$$A \models_F \varphi \iff C \models_F \varphi \ .$$

Proof. The proof follows exactly that of Theorem 3. Since $F \subseteq M'$ and all transitions corresponding to executions of methods in M' are taken for the construction of stuttering equivalent paths, we will always construct fair paths when starting from a fair path. □

6. Conclusion

This work is concerned with the re-use of verification results of classes. Given a verified class the technique presented in this paper can be used to determine whether some specific property is preserved under a change made to the class. The technique relies on the representation of the dependencies of a class specification in a program dependence graph. On this graph it is possible to determine the effect of changes on the behaviour of a class. As a special case we looked at changes inducing behavioural subtypes in which all properties (talking about the original class) are preserved.

So far, this technique considers a single class only. It could be extended to larger systems either by combining it with compositional verification techniques (e.g. for Object-Z [31]), or by constructing a program dependence graph of the whole system. The latter could be achieved by combining program dependence graphs of the individual objects through a special new dependency arc reflecting the call structure between objects (possibly following approaches for slicing programs with procedures).

6.1. Related work

The basic technique that we use for describing the dependencies between entities in a specification is a standard technique in program analysis and useful for answering many different types of questions [12]. The predominant method using dependence graphs is

slicing. Slicing [29,30] was introduced to facilitate debugging of programs: a programmer should be presented with just that part of the program that potentially influences the errors he/she is currently trying to debug. While first approaches to slicing only treated simple imperative programs without procedures the technique was soon extended to programs with procedures, pointers, concurrency etc. (see [24] for an overview).

The use of slicing techniques (or at least, related ideas) in verification have first appeared in hardware verification where a technique called *cone-of-influence reduction* was developed to reduce circuit models before verifying their correctness [4]. Slicing in software verification has in particular been used for verifying Java programs (Bandera project [8,9]), and recently also for Promela, the input language of the SPIN modelchecker [18]. For Java, slicing is performed with respect to temporal logic properties as well, however, with the aim of *reducing* the program to be checked, not for determining the impact of changes on the validity of a formula.

In the area of program verification of object-oriented programs inheritance of properties to derived classes, or more specific behavioural subtypes, is intensively studied as well. Leavens and Weihl [14] show how to verify object-oriented programs using a technique called "supertype abstraction". This technique is based on the idea that subtypes need not to be re-verified once a property has been proven for their supertypes. In their study they have to take particular care about *aliasing* since in object-oriented programs several references may point to the same object, and thus an object may be manipulated in several ways. Subtyping for object-oriented programs has to avoid references which are local to the supertype but accessible in the subtype. Alagic and Kouznetsova [1] study behavioural compatibility in the presence of self-typing, i.e. where formulae of a logic may refer to the particular type of an object itself. The general aim of these works is to give precise conditions for when properties of classes are inherited to derived classes. The main difference to our work lies in the properties treated (here expressed in temporal logic) and in the language or formalism under consideration. While we use a *specification language* the before mentioned approaches deal with object-oriented programming languages where specific issues such as object references, aliasing and polymorphism play an important role. The version of Object-Z that we use here (which is the Object-Z part of CSP-OZ [6]) does not include object references. Instead, communication between objects is done in a CSP-like style by sending messages over channels. The link to an actual implementation in an object-oriented programming language is achieved by generating *assertions* on (Java) programs [19], which are checked at runtime.

The issue of inheritance of properties to subtypes in the area of Petri nets has been treated by van der Aalst and Basten [25]. They deal with net-specific properties like safety (of nets), deadlock freedom and free choice.

Preservation of properties is also an issue in transformations within the language UNITY proposed by Chandy and Misra [3]. The *superposition* operator in UNITY is a form of parallel composition which requires that the new part does not make assignments to underlying (old) variables. Superposition preserves all properties of the original program.

Another area of related work is the field of *change impact analysis* in software engineering (see for instance [15]). There, similar techniques (dependence graphs) are employed to find out what the effects of changes on the software are. Particular properties, like those treated here, are not in the focus of change impact analysis, rather it is used to determine the entities

(e.g. classes or objects) in the software which might be affected by a change. An approach which is in spirit similar to ours can be found in the area of testing: Regression testing is concerned with analysing the impact of changes on tests in order to determine which tests have to be re-run. Regression testing also employs slicing techniques (see for instance [7]).

6.2. Future work

In the future we intend to extend the technique presented here to integrated specification methods covering—beside data and operations as presented here—also *process description* and *timing constraints*. The formalism we are aiming at is the specification technique CSP-OZ-DC [11] already mentioned in the introduction. While the dependencies arising from processes describing the ordering of method invocations can still be tackled with more or less standard concepts (control dependencies), timing constraints pose new questions as they give rise to a new kind of dependency. Moreover, the underlying semantic domain will then deviate substantially from our current semantic domain of Kripke structure and will thus necessitate more complex proofs.

Another direction of extension lies in the granularity of the dependence graph. Currently, the dependence graph uses schemas as nodes. To improve the effect of slicing, i.e. possibly find more variables which could be removed, the level of granularity could be moved to *predicates* within schemas. This idea is elaborated on in [2] with the intent of *reducing* an Object-Z specification with respect to certain properties under interest.

Acknowledgements

Many thanks to the two anonymous referees who pointed to interesting related work and gave detailed comments that helped to improve the paper.

References

[1] S. Alagic, S. Kouznetsova, Behavioral compatibility of self-typed theories, in: Boris Magnusson (Ed.), ECOOP 2002—Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings, Vol. 2374, Springer, Berlin, June 2002, pp. 585–608.

[2] I. Brückner, H. Wehrheim, Slicing Object-Z Specifications for Verification, in: H. Treharne, S. King, M. Henson, S. Schneider (Eds.), ZB2005: Formal Specification and Development in Z and B, LNCS 3455, Springer, Berlin, 2005, pp. 414–434.

[3] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, Reading, MA, 1988.

[4] E. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, Cambridge, MA, 1999.

[5] R. Duke, G. Rose, G. Smith, Object-Z: A specification language advocated for the description of standards, Comput. Standards Interfaces 17 (1995) 511–533.

[6] C. Fischer, CSP-OZ: A combination of Object-Z and CSP, in: H. Bowman, J. Derrick (Eds.), Formal Methods for Open Object-Based Distributed Systems (FMOODS '97), Vol. 2, Chapman & Hall, London, 1997, pp. 423–438.

[7] R. Gupta, M. Harrold, M. Soffa, An approach to regression testing using slicing, in: Proc. of the International Conference on Software Maintenance, 1992, pp. 299–308.

[8] J. Hatcliff, M. Dwyer, Using the Bandera tool set to model-check properties of concurrent Java software, in: K.G. Larsen (Ed.), CONCUR 2001, Lecture Notes in Computer Science, Springer, Berlin, 2001.

[9] J. Hatcliff, M. Dwyer, H. Zheng, Slicing software for model construction, Higher-order Symbolic Comput. 13 (4) (2000) 315–353.
[10] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
[11] J. Hoenicke, E.-R. Olderog, CSP-OZ-DC: A combination of specification techniques for processes data and time, Nordic J. Comput. 9 (4) (2003) 301–334.
[12] S. Horwitz, T.W. Reps, The use of program dependence graphs in software engineering, in: Internat. Conf. on Software Engineering, 1992, pp. 392–411.
[13] K. Huizing, R. Kuiper, Reinforcing fragile base classes, in: A. Poetzsch-Heffter (Ed.), Workshop on Formal Techniques for Java Programs ECOOP, New 2001, 2001.
[14] G.T. Leavens, W.E. Weihl, Specification and verification of object-oriented programs using supertype abstraction, Acta Inform. 32 (1995) 705–778.
[15] M.L. Lee, Change impact analysis of object-oriented software, Ph.D. Thesis, George Mason University, 1998.
[16] B. Liskov, J. Wing, A behavioural notion of subtyping, ACM Trans. Programming Languages Syst. 16 (6) (1994) 1811–1841.
[17] Z. Manna, A. Pnueli, The temporal logic of reactive and concurrent systems (Specification), Springer, 1991.
[18] L. Millett, T. Teitelbaum, Issues in slicing promela and its applications to model checking, protocol understanding, and simulation, Software Tools Technol. Transfer 2 (4) (2000) 343–349.
[19] M. Möller, E.R. Olderog, H. Rasch, H. Wehrheim, Linking CSP-OZ with UML and Java: A Case Study, in: Integrated Formal Methods, no. 2999 in Lecture Notes in Computer Science, Springer, March 2004, pp. 267–286.
[20] D. Peled, T. Wilke, Stutter-invariant temporal properties are expressible without the next-time operator, Inform. Process. Lett. 63 (5) (1997) 243–246.
[21] A. Poetzsch-Heffter, J. Meyer, Interactive verification environments for object-oriented languages, J. Universal Comput. Sci. 5 (3) (1999) 208–225.
[22] G. Smith, The Object-Z Specification Language, Kluwer Academic Publishers, Dordrecht, 2000.
[23] G. Smith, K. Winter, Proving Temporal Properties of Z Specifications Using Abstraction, in: D. Bert, J.P. Bowen, S. King, M. Walden (Eds.), ZB 2003: Formal Specification and Development in Z and B, no. 2651 in Lecture Notes in Computer Science, Springer, Berlin, 2003, pp. 260–279.
[24] F. Tip, A survey of program slicing techniques, J. Program. Languages 3 (3) (1995).
[25] W.M.P. van der Aalst, T. Basten, Inheritance of Workflows—An approach to tackling problems related to change, Theoret. Comput. Sci. 270 (1–2) (2002) 125–203.
[26] H. Wehrheim, Behavioural subtyping and property preservation, in: S. Smith, C. Talcott (Eds.), FMOODS'00: Formal Methods for Open Object-Based Distributed Systems, Kluwer, Dordrecht, 2000.
[27] H. Wehrheim, Inheritance of temporal logic properties, in: P. Stevens, U. Nestmann (Eds.), FMOODS 2003: Formal Methods for Open Object-based Distributed Systems, Vol. 2884. of Lecture Notes in Computer Science, Springer, Berlin, 2003, pp. 79–93.
[28] H. Wehrheim, Preserving Properties under Change, in: F.S. de Boer, M.M. Bonsague, S. Graf, W.-P. de Roever (Eds.), FMCO 2003: Formal Methods for Components and Objects, Lecture Notes in Computer Science, 3188, Springer, Berlin, 2004, pp. 330–343.
[29] M. Weiser, Programmers use slices when debugging, Comm. ACM 25 (7) (1982) 446–452.
[30] M. Weiser, Program slicing, in: Proc. of the Fifth International Conference on Software Engineering, IEEE Press, 1981, pp. 439–449.
[31] K. Winter, G. Smith, Compositional Verification for Object-Z, in: D. Bert, J.P. Bowen, S. King, M. Walden (Eds.), ZB 2003: Formal Specification and Development in Z and B, no. 2651 in Lecture Notes in Computer Science, Springer, Berlin, 2003, pp. 280–299.
[32] Z. Chaochen, C.A.R. Hoare, A.P. Ravn, A calculus of durations, Inform. Process. Lett. 40/5 (1991) 269–276.